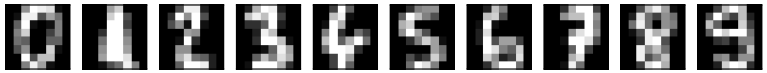
K-plus proches voisins

1 Introduction

1.1 Objectifs poursuivis

Le but de cette séance de travaux pratiques est essentiellement de reprendre les idées vu en cours. Le stockage et la représentation des données différeront très légèrement, mais le principe du classifieur est le même. On cherche cette fois à effectuer de la reconnaissance d'écriture, et plus précisément celle de chiffres (de 0 à 9) écrits à la main, une tache fréquemment utile (par exemple, pour la lecture automatique du code postal à vu de tri automatisé du courrier). Le but recherché est que le classifieur, lorsqu'on lui présente une image contenant un chiffre, soit capable de l'identifier avec robustesse.

On travaillera avec une base de chiffres appelée MNIST. Dans la version sur laquelle nous travaillerons, chaque chiffre a été prétraité : les données sont des imagettes noir et blanc de taille 8×8 où le chiffre est centré et redimensionné pour que tous les chiffres aient la même taille, comme illustré ci-dessous. Une opération préalable indispensable pour de bons résultats de classification.



1.2 Chargement des données

On charge la base de données avec :

```
from sklearn import datasets
digits = datasets.load_digits()
```

Dans l'objet nommé `digits` ainsi obtenu, on trouve plusieurs listes de taille N, N étant le nombre de chiffres manuscrits présents dans la base, dont les listes suivantes :

- `digits.data` contient les données qui serviront à la reconnaissance : il s'agit de tableaux de longueur 64 contenant des valeurs de 0 à 16 représentant l'intensité des différents pixels ;
- `digits.images` contient les images correspondantes (en fait, il s'agit des *mêmes* données que `digits.data`, mais, par confort, sous la forme de tableaux 8×8 plutôt que de tableaux unidimensionnels [1] ;
- `digits.target` contient la "vérité" concernant les chiffres, sous la forme d'un entier entre 0 et 9, autrement dit `digits.target[i]` contient le chiffre correspondant à `digits.data[i]`.

1. En réalité, cet apparent double stockage ne nécessite pas un stockage séparé dans la mémoire.

Pour afficher un chiffre à partir de ses données images, on peut écrire :

```
import matplotlib.pyplot
plt.imshow(digits.images[i], cmap="gray")
plt.show()
```

1. Afficher quelques imagettes de chiffres (il y a 1797 exemples dans la base de données) et vérifier que vous arrivez vous-même à les reconnaître (on vérifiera avec `data.target[i]`)

Dans la suite, on ne travaillera qu'avec les données de la liste `data` (des tableaux de longueur 64). On peut toutefois quand même afficher un chiffre à partir d'un tableau `d` de taille 64 en écrivant :

```
plt.imshow(d.reshape((8,8)), cmap="gray")
plt.show()
```

2 Classification

2.1 Prétraitements

On souhaite tout d'abord, dans un premier temps, réorganiser les données pour faciliter les traitements par la suite.

1. Construire un dictionnaire `data` dont les clés sont les entiers de 0 à 9, et les valeurs associées des listes contenant les tableaux de longueur 64 associés aux chiffres de la base de données. On pourra partir de l'ébauche suivante :

```
data = {}  # Un dictionnaire initialement vide

for i in range(10):
    data[i] = []     # Une liste vide pour chaque clé (chiffre)

for i in range(len(digits.target)):
    data[ ... ].append( ... )
```

On vérifiera que l'on a le bon nombre d'éléments par classe :

```
>>> [len(data[i]) for i in range(10)]
[178, 182, 177, 183, 181, 182, 181, 179, 174, 180]
```

On notera ici que l'on dispose d'au moins 170 exemples dans chaque « classe ».

Et contrôlera également que les objets sont bien des tableaux de taille 64 :

```
>>> len(data[0][0])
64
```

Et que leur contenu est le bon :

```
>> data[0][0]
array([ 0.,  0.,  5., 13.,  9.,  1., ...

>> data[0][177]
array([ 0.,  0.,  6., 16., 13., 11., ...
```

L'étape suivante consiste à construire un groupe de données qui serviront de représentants pour l'algorithme des k plus proches voisins, et un groupe de données qui serviront à tester les résultats.

On souhaite avoir 30 exemples pour chaque chiffre comme référence, 140 pour chaque chiffre comme tests. En théorie, il conviendrait de les sélectionner au hasard. Pour faciliter les échanges entre vous (et avoir les mêmes résultats), nous allons exceptionnellement omettre cette étape [2] (les exemples de la base ne sont de toute façon pas rangés dans un ordre particulier). Ainsi, pour chaque classe :

- les 30 premiers chiffres serviront de « références » ;
- les 140 chiffres suivants serviront pour nos tests ;
- les quelques chiffres restants seront ignorés.

On rappelle que `L[20:80]` permet d'obtenir une liste contenant les éléments d'index 20 à 79 de la liste L.

2. Construire un dictionnaire `ref` dont les clés sont les entiers de 0 à 9 et tel que `ref[i]` contienne les 30 premiers tableaux de taille 64 de `data[i]` (ceux indexés de 0 à 29).

3. Construire de même un dictionnaire `test` dont les clés sont les entiers de 0 à 9 et tel que `test[i]` contienne les 140 tableaux de taille 64 suivants de `data[i]` (ceux indexés de 30 à 169).

On vérifiera que l'on a le bon nombre d'éléments par classe :

```
>>> [len(ref[i]) for i in range(10)]
[30, 30, 30, 30, 30, 30, 30, 30, 30, 30]
```

Il doit en être de même pour `test`, avec cette fois 140 données par classe.

2. Cela resterait en principe possible même avec un mélange avec les fonctions pseudo-aléatoires dont on dispose en forçant la graîne aléatoire, mais c'est plus délicat à garantir.

On peut également contrôler que le contenu de ces listes est bon avec ces deux tests :

```
>>> hash(tuple(map(lambda x:tuple(map(tuple, x)),
                                        (ref[i] for i in range(10)))))
-8181673141168343966

>>> hash(tuple(map(lambda x:tuple(map(tuple, x)),
                                       (test[i] for i in range(10)))))
-4633577574915266473
```

Si les valeurs sont différentes, il est probable que vous n'ayez pas choisi les bonnes parties de chaque liste.

2.2 Définition de la distance

Pour mesurer la distance entre deux objets, on utilisera simplement la distance euclidienne entre les intensités de leurs pixels (codage rétinien).

4. Proposer une fonction `dist2(t1, t2)` retournant un flottant représentant le carré de la distance euclidienne entre les données du tableau `t1` et celles du tableau `t2`.

On pourra tester la fonction précédente de la sorte (on remarquera au passage que deux « 0 » sont plus proches qu'un « 0 » et un « 1 ») :

```
>>> dist2(ref[0][0], ref[0][1])
562.0

>>> dist2(ref[0][0], ref[1][0])
3547.0
```

2.3 Calcul des distances

Pour une donnée quelconque (un tableau de taille 64), on souhaite déterminer sa distance à chacun des 300 chiffres « référence ».

5. Proposer une fonction `toutes_distances(d, ref)` qui prend en argument un tableau `d` de longueur 64 et un dictionnaire `ref` contenant les 300 chiffres de référence, et retournant une liste de 300 couples `(d2, k)` où `d2` représente le carré distance euclidienne à un des chiffres de référence, et `k` est l'entier (entre 0 et 9 correspondant à ce chiffre.

6. Modifier la fonction précédente pour que la liste retournée soit triée par ordre croissant des distances. On remarquera que comme les éléments sont des couples qui commencent par la distance, et que Python compare des couples en se basant en priorité sur leur premier élément (ordre lexicographique), on peut utiliser une comparaison directe sur les couples ici. On pourra utiliser `.sort()` pour ne pas perdre de temps, mais gardez en tête qu'il faut être capable d'écrire au moins un tri, même élémentaire, aux concours.

On pourra tester la fonction de la façon suivante :

```
>>> toutes_distances(test[0][0], ref)[:5]
[(405.0, 0), (436.0, 0), (437.0, 0), (519.0, 0), (667.0, 0)]

>>> toutes_distances(test[9][5], ref)[:5]
[(1113.0, 5), (1147.0, 9), (1153.0, 9), (1355.0, 7), (1389.0, 5)]
```

On remarquera donc que le sixième « neuf » dans l'ensemble de test n'est pas facile à ne pas confondre avec un 5 ou un 7, et on pourra afficher l'imagette pour comprendre pourquoi ça n'a rien de surprenant :

```
plt.imshow(test[9][5].reshape((8,8)), cmap="gray")
plt.show()
```

2.4 Classifieur et analyse des résultats

7. Écrire une fonction `gagnant(lst)` prenant en argument une liste d'entiers entre 0 et 9, et retournant l'entier apparaissant le plus grand nombre de fois dans cette liste (en cas d'égalité, on retournera, parmi les ex-aequos, celui qui apparaît en premier dans la liste). On s'efforcera d'obtenir une complexité linéaire en la taille de la liste.

8. En s'appuyant sur les fonctions `toutes_distances` et `gagnant`, proposer une fonction `kppv(d, ref, k)` prenant en argument une donnée `d` (tableau de longueur 64) et retournant l'entier entre 0 et 9 correspondant à une classification avec l'algorithme des k plus proches voisins sur les données de référence `ref`. On rappelle que si `L` est une liste, on peut obtenir la liste des `k` premiers éléments de `L` en écrivant `L[:k]`.

9. Sur les 1000 chiffres de notre ensemble de tests, combien sont bien classés pour $k = 1$? $k = 7$?

10. Construire la matrice de confusion (de taille 10×10) de notre classifieur pour $k = 7$. Quels sont les chiffres les plus souvent confondus ?

2.5 Choix de k

Utiliser un paramètre k élevé est intéressant lorsqu'il y a des « outliers » dans les données, c'est à dire des « intrus » au beau milieu d'un classe, comme on l'a vu avec les iris. En fait, il y en a relativement peu avec la base MNIST que l'on manipule, de sorte que $k = 1$ serait souvent la meilleure solution.

Pour montrer l'intérêt de k pour résister à ces intrus, nous allons dégrader la base de référence en classant volontairement incorrectement certains chiffres :

```
ref_degr = {
    i: ref[i][:20] + [ref[(i+k)%10][20+k] for k in range(10)]
    for i in range(10) }
```

Il y a toujours 30 exemples par classe mais dorénavant, 9 sont en fait des chiffres d'une *autre* classe. Si l'on fait tourner les fonctions précédentes sur cette base dégradée, évidemment la classification sera moins bonne.

Dans la suite, on travaillera avec `ref_degr` comme base de référence plutôt qu'avec `ref`.

On souhaite déterminer l'évolution du nombre de chiffres bien classés en fonction de k sur cette base dégradée. Il est possible d'utiliser les fonctions précédentes, mais cela prendrait un peu de temps. On va procéder un peu différemment.

11. Proposer une fonction `majoritaire(lst)` prenant en argument une liste d'entiers `lst` et retournant une liste `res` de même longueur, telle que `res[i]` contienne l'entier le plus représenté dans `lst` aux positions d'indice `0` à `i` inclus (en cas d'égalité, on choisira l'entier qui apparaît en premier).

12. En déduire une fonction `classification(d, ref)` qui prend en argument un tableau de longueur 64 et un dictionnaire contenant les 300 chiffres de référence, et retournant une liste `res` de 300 entiers telle que `res[i]` contienne le résultat de la classification par k plus proches voisins pour $k = i + 1$.

13. Construire une liste `resultats` de 1000 listes de 300 éléments, contenant les listes construites par la fonction précédente pour les 1000 exemples de notre ensemble de test. Les 100 premiers éléments correspondront aux imagettes de zéros, les 100 suivants aux imagettes de uns, etc. On pensera bien à utiliser `ref_degr` comme base de référence (on pourra éventuellement comparer les résultats avec la base originale, si le temps le permet).

La construction de cette liste prend du temps, on s'efforcera de préserver la liste ainsi construite pour la suite !

14. Proposer une fonction `taux_succes(resultats, k)` qui retourne le pourcentage de classement correct en fonction de k (k étant un entier entre 1 et 700).

15. Tracer la courbe du taux de succès en fonction de `k`. Quel `k` peut-on choisir ?

16. Proposer une fonction `matrice_confusion(resultats, k)` retournant une liste `conf` de 10 listes contenant 10 entiers, telle que `conf[i][j]` contienne le nombre d'imagettes correspondant au chiffre `i` reconnues comme le chiffre `j` pour la méthode des k plus proches voisins (k étant un entier entre 1 et 300)

17. Quels sont les chiffres les plus souvent confondus pour le k choisi ?